# Introduction au langage C

## Petite histoire du C

Le langage C a été mis au point par **D.Ritchie** et **B.W.Kernighan** au début des années 70. Leur but était de permettre de développer un langage qui permettrait d'obtenir un système d'exploitation de type UNIX portable. D.Ritchie et B.W.Kernighan se sont inspirés des langages B et BCPL, pour créer un nouveau langage : le langage C.

La première définition de ce langage a été donnée dans leur livre commun « The C programming language ».
Toutefois, suite à l'apparition de nombreux compilateurs C, l'ANSI (abréviation de *American National Standards Institute*) a décidé de normaliser ce langage pour donner ce que l'on appelle le **C-ANSI**. Suite à cette norme, Ritchie et Kernighan ont sorti une deuxième édition du livre en intégrant les modifications apportées par l'ANSI.

## Les atouts du C

Le langage C reste un des langages les plus utilisés actuellement. Cela est dû au fait que le langage C est un langage comportant des instructions et des structures de haut niveau (contrairement à l'assembleur par exemple) tout en générant un code très rapide grâce à un compilateur très performant.

Un des principaux intérêts du C est que c'est un langage très **portable**. Un programme écrit en C en respectant la norme ANSI est portable sans modifications sur n'importe quel système d'exploitation disposant d'un compilateur C : Windows, UNIX, VMS (système des VAX) ou encore OS/390 ou z/Os (l'OS des mainframes IBM).

La rapidité des programmes écrits en C est en grande partie due au fait que le compilateur *présuppose* que le programmeur sait ce qu'il fait : il génère un code ne contenant pas de vérifications sur la validité des pointeurs, l'espace d'adressage, etc.
Ainsi, les programmes en C sont très compacts.

De plus, une des caractéristiques du C est qu'il est un langage « faiblement typé » : les types de données qu'il manipule sont très restreints, et proches de la représentation interne par le processeur : par exemple, le type 'Chaîne de caractères' n'existe pas en C. A l'inverse, comparer un entier et un caractère a un sens en C car un caractère est bien représenté en interne par le processeur par une valeur de type entier (le code ASCII ou le code EBCDIC).

Enfin et pour conclure, il est inexact que le C est un langage difficile à apprendre ! Au contraire : le C dispose de peu d'instructions, les structures de données sont limitées, etc. Le C est un langage concis et son apprentissage est beaucoup moins ardu que ne peut l'être celui du Pascal par exemple.

L'apprentissage du C est ainsi nécessaire pour quiconque s'intéresse à la programmation, et cet apprentissage en vaut la peine !

## Le C++ par rapport au C

Le C++ est un langage basé sur le langage C, auquel on a rajouté des éléments de telle manière à intégrer le concept objet. C'est **Bjarne Stroustrup** qui a créé la première version de ce langage, appelé C++.

# Caractéristiques du langage C

---

## Le fichier source

Le fichier source d'un programme écrit en langage C est un simple fichier texte dont l'extension est par convention *.c*.



**Note**:
L'extension est en minuscules. Le .C (majuscule) est interprété par certains compilateurs comme l'extension du C++ (gcc). Comme il existe de petites différences entre la compilation d'un programme en C et la compilation de ce même programme en C++, cela peut parfois poser des problèmes.

Ce fichier source doit être un fichier texte non formaté, c'est-à-dire un fichier texte dans sa plus simple expression, sans mise en forme particulière ou caractères spéciaux (il contient uniquement les caractères ASCII de base).

Lorsque le programme est prêt à être « essayé », il s'agit de le compiler (le traduire en langage machine).
De nombreux compilateurs C existent : sous les systèmes de type UNIX par exemple, le compilateur C est fourni en standard, si bien que la programmation en langage C est aisée sous ce type de système. La compilation sous UNIX se fait par la ligne de commande suivante :

```
cc fichier.c
```

## Aspect d'un programme en C

Un programme écrit en langage C comporte une fonction principale appelée *main()* renfermant les instructions qui doivent être exécutées. Celles-ci sont comprises entre des accolades qui suivent le nom de la fonction. Cela vous semble tombé du ciel si vous n'avez jamais programmé en C, mais il faut admettre pour l'instant la manière d'écrire un programme en C. La finalité de cette écriture vous sera dévoilée au cours des chapitres suivants...

Un programme C de base ressemblera donc à ceci :

```
main()

{

        printf("Ceci est votre premier programme");

}
```

Le programme présenté ci-dessus contient donc une fonction principale *main()* (qui, rappelons-le, est essentielle car c'est par cette fonction que le programme s'exécute) contenant une instruction imprimant à l'écran le message « Ceci est votre premier programme » grâce à la fonction *printf()*.

**Note**:

1. Le type retourné par main() est int. La norme actuelle du C (C99) impose que le type soit explicite, il faut donc écrire :
2. 
3. `int main()`
4. 
5. Il est recommandé de définir une fonction sous sa forme prototypée. Dans ce cas, main() n'ayant pas de paramètres, on l'indique avec void.
6. 
7. `int main(void)`
8. 
9. printf() est une fonction avec un nombre variable de paramètres. Il est obligatoire de fournir un prototype à cette fonction. Il manque par exemple :
10. 
11. `#include <stdio.h>`
12. 

13. *printf()* produit une émission de caractères en séquence vers *stdout*. Certaines implémentations de *stdout* étant bufférisées, il est recommandé de terminer la chaîne émise par un '*n*', ce qui déclenche l'émission effective. Sinon, il est possible de la forcer avec *fflush(stdout)* :

```
 printf ("Ceci est votre premier programme\n");
```

17. Bien que la norme actuelle (C99) autorise main() à ne pas avoir de *return* explicite (dans ce cas on a un return 0 implicite), cette pratique est peu recommandée pour des questions de compatibilité avec la norme courante (C90) qui exige qu'une fonction retournant autre chose que void ait un return quelque chose explicite.

```
 return 0;
```

Je propose :

```
#include <stdio.h>



int main (void)

{

printf ("Ceci est votre premier programme\n");

return 0;

}
```

## Typologie

La manière d'écrire les choses en langage C a son importance. Le langage C est par exemple sensible à la casse (en anglais *case sensitive*), cela signifie qu'un nom contenant des majuscules est différent du même nom écrit en minuscules. Ainsi, les spécifications du langage C précisent que la fonction principale doit être appelée *main()* et non *Main()* ou *MAIN()*.

De la même façon, on remarquera que la fonction *printf()* est écrite en minuscules. D'autre part, l'instruction *printf()* se termine par un point-virgule. Ce détail a son importance, car en langage C, **toute instruction se termine par un point-virgule**.

## Ajout de commentaires

Lorsqu'un programme devient long et compliqué, il peut être intéressant (il est même conseillé) d'ajouter des lignes de commentaires dans le programme, c'est-à-dire des portions du fichier source qui ont pour but d'expliquer le fonctionnement du programme sans que le compilateur ne les prenne en compte (car il générerait une erreur).

Pour ce faire, il faut utiliser des balises qui vont permettre de délimiter les explications afin que le compilateur les ignore et passe directement à la suite du fichier.
Ces délimiteurs sont /* et */. Un commentaire sera donc noté de la façon suivante :

```
/*Voici un commentaire !*/
```

En plus des symboles /* et */, fonctionnant un peu comme des parenthèses, le symbole // permet de mettre en commentaire toute la ligne qui la suit (i.e. les caractères à droite de ce symbole sur la même ligne).

Il convient toutefois d'utiliser préférablement la notation /* */ que //, car c'est beaucoup plus joli et plus propre. La notation // est généralement réservée pour mettre en commentaire une ligne de code que l'on souhaite désactiver temporairement.

Il y a toutefois quelques règles à respecter :



- Les commentaires peuvent être placés n'importe où dans le fichier source
- Les commentaires ne peuvent contenir le délimiteur de fin de commentaire (*/)
- Les commentaires ne peuvent être imbriqués
- Les commentaires peuvent être écrits sur plusieurs lignes
- Les commentaires ne peuvent pas couper un mot du programme en deux

# Notion de processeur

## Définition du préprocesseur

Dans les chapitres précédents, un programme simple vous a été présenté, il s'agit du programme suivant :

```
int main(void)

{

        printf("Ceci est votre premier programme");

}
```

Dans ce programme la fonction principale *main()* contient une fonction appelée *printf()* qui a pour but d'afficher le message « Ceci est votre premier programme ». En réalité le compilateur ne connaît pas la fonction *printf()* bien qu'il s'agisse d'une fonction standard du langage C. Cette fonction est effectivement stockée dans un fichier annexe, contenant une librairie de fonctions, appelé **fichier de définition** (éventuellement **fichier d'en-tête** ou **fichier header**), dont l'extension est *.h*.
Il s'agit donc de préciser au compilateur dans quel fichier se trouve la définition de la fonction *printf()*...

Celle-ci se trouve dans le fichier d'en-tête appelé *stdio.h*.
La désignation *stdio* signifie *Standard Input Output* (en français *Entrées Sorties standard*), c'est-à-dire que ce fichier contient la définition de nombreuses fonctions permettant l'entrée (saisie au clavier, ...) et la sortie (affichage, sortie dans un fichier, ...) de données, dont la fonction *printf()* fait partie.

L'incorporation de la déclaration de la fonction *printf()* se fait au moyen de l'instruction *#include* (que l'on place en début de fichier) suivie des balises < et > contenant le nom de fichier contenant la définition de la fonction. La déclaration *include* doit se trouver avant toute utilisation des méthodes déclarées, sinon le compilateur générera au minimum un warning.
Le programme ci-dessus pour pouvoir être compilé doit donc s'écrire :

```
#include <stdio.h>



int main(void)

{

        printf("Ceci est votre premier programme");

}
```

Le fichier est maintenant apte à être compilé. Il existe d'autres commandes du préprocesseur qui seront détaillées dans ce cours.



Les instructions dédiées au préprocesseur ne se terminent **pas** par un point-virgule !

### Phases de compilation

La compilation se fait généralement en plusieurs phases :




- le compilateur transforme le code source en code objet, et le sauvegarde dans un **fichier objet**, c'est-à-dire qu'il traduit le fichier source en langage machine (certains compilateurs créent aussi un fichier en assembleur, un langage proche du langage machine car possédant des fonctions très simples, mais lisible).
- le compilateur fait ensuite appel à un **éditeur de liens** (en anglais **linker** ou **binder**) qui permet d'intégrer dans le fichier final tous les éléments annexes (fonctions ou librairies) auquel le programme fait référence mais qui ne sont pas stockés dans le fichier source.
  Puis il crée un **fichier exécutable** qui contient tout ce dont il a besoin pour fonctionner de façon autonome, le fichier ainsi créé possède l'extension .exe.

Dans le cas du langage C, une phase supplémentaire apparaît, il s'agit du traitement du fichier par le préprocesseur C, un programme permet d'inclure dans le fichier source les éléments référencés par les instructions situées au début du fichier source (instructions précédées du caractère #). C'est donc le préprocesseur qui ajoutera dans le fichier source la définition de la fonction *printf()* qu'il sera allé chercher dans le fichier *stdio.h* grâce à l'instruction *#include*.

Les phases de compilation dans le cas d'un compilateur C sont donc les suivantes :




# Types de données

### Les types de données

Les données manipulées en langage C sont typées, c'est-à-dire que pour chaque donnée que l'on utilise (dans les variables par exemple) il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation :

- **des nombres** : entiers (**int**) ou **réels**, c'est-à-dire à virgules (**float**)
- **des pointeurs** (pointer) : permettent de stocker l'adresse d'une autre donnée, ils « pointent » vers une autre donnée

En C il existe plusieurs types entiers, dépendant du nombre d'octets sur lesquels ils sont codés ainsi que de leur format, c'est-à-dire s'ils sont signés (possédant le signe - ou +) ou non. Par défaut les données sont signées.

Voici un tableau donnant les types de données en langage C :

| Type de donnée | Signification | Taille (en octets) | Plage de valeurs acceptée |
|---|---|---|---|
| char | Caractère | 1 | -128 à 127 |

| | | | |
|---|---|---|---|
| unsigned char | Caractère non signé | 1 | 0 à 255 |
| short int | Entier court | 2 | -32 768 à 32 767 |
| unsigned short int | Entier court non signé | 2 | 0 à 65 535 |
| int | Entier | 2 (sur processeur 16 bits)<br>4 (sur processeur 32 bits) | -32 768 à 32 767<br>-2 147 483 648 à 2 147 483 647 |
| unsigned int | Entier non signé | 2 (sur processeur 16 bits)<br>4 (sur processeur 32 bits) | 0 à 65 535<br>0 à 4 294 967 295 |
| long int | Entier long | 4 | -2 147 483 648 à 2 147 483 647 |
| unsigned long int | Entier long non signé | 4 | 0 à 4 294 967 295 |
| float | Flottant (réel) | 4 | $3.4*10^{-38}$ à $3.4*10^{38}$ |
| double | Flottant double | 8 | $1.7*10^{-308}$ à $1.7*10^{308}$ |
| long double | Flottant double long | 10 | $3.4*10^{-4932}$ à $3.4*10^{4932}$ |

## Nombre entier (*int*)

Un nombre entier est un nombre sans virgule qui peut être exprimé dans différentes bases:

- Base décimale : l'entier est représenté par une suite de chiffres unitaires (de 0 à 9) ne devant pas commencer par le chiffre 0
- Base hexadécimale : l'entier est représenté par une suite d'unités (de 0 à 9 ou de A à F (ou a à f)) devant commencer par *0x* ou *0X*
- Base octale : l'entier est représenté par une suite d'unités (incluant uniquement des chiffres de 0 à 7) devant commencer par *0*

Les entiers sont signés par défaut, cela signifie qu'ils comportent un signe. Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le complément à deux.

## Nombre à virgule (*float*)

Un nombre à virgule flottante est un nombre à virgule, il peut toutefois être représenté de différentes façons :

- un entier décimal : 895
- un nombre comportant un point (et non une virgule) : 845.32
- une fraction : 27/11
- un nombre exponentiel, c'est-à-dire un nombre (éventuellement à virgule) suivi de la lettre *e* (ou *E*), puis d'un entier correspondant à la puissance de 10 (signé ou non, c'est-à-dire précédé d'un + ou d'un -) :
  2.75e-2
  35.8E+10
  .25e-2

En réalité, les nombres réels sont des nombres à virgule flottante, c'est-à-dire un nombre dans lequel la position de la virgule n'est pas fixe, et est repérée par une partie de ses bits (appelée l'exposant), le reste des bits permettent de coder le nombre sans virgule (la mantisse).

Les nombres de type **float** sont codés sur 32 bits dont :

- 23 bits pour la mantisse
- 8 bits pour l'exposant
- 1 bit pour le signe

Les nombres de type **double** sont codés sur 64 bits dont :

- 52 bits pour la mantisse
- 11 bits pour l'exposant
- 1 bit pour le signe

Les nombres de type **long double** sont codés sur 80 bits dont :

- 64 bits pour la mantisse
- 15 bits pour l'exposant
- 1 bit pour le signe

La précision des nombres réels est approchée. Elle dépend par le nombre de positions décimales, suivant le type de réel elle sera au moins :

- de 6 chiffres pour le type **float**
- de 15 chiffres pour le type **double**
- de 17 chiffres pour le type **long double**

### Caractère (*char*)

Le type **char** (provenant de l'anglais *character*) permet de stocker la valeur ASCII d'un caractère, c'est-à-dire un nombre entier !

Par défaut les nombres sont signés, cela signifie qu'ils comportent un signe. Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le complément à deux. Une donnée de type char est donc signée, cela ne signifie bien sûr pas que la lettre possède un signe mais tout simplement que dans la mémoire la valeur codant le caractère peut être négative...

Si jamais on désire par exemple stocker la lettre B (son code ASCII est 66), on pourra définir cette donnée soit par le nombre 66, soit en notant 'B' où les apostrophes simples signifient *code ascii de...*

Il n'existe pas de type de données pour les chaînes de caractères (suite de caractères) en langage C. Pour créer une chaîne de caractères on utilisera donc des tableaux contenant dans chacune de ses cases un caractère...

## Créer un type de donnée

Il est possible en C de définir un nouveau type de données grâce au mot clé *typedef*. Celui-ci admet la syntaxe suivante :

```
typedef Caracteristiques_du_type Nom_du_type
```

où

- *Caracteristiques_du_type* représente un type de données existant (par exemple *float*, *short int*, ...)
- *Nom_du_type* définit le nom que vous donnez au nouveau type de donnée

Ainsi l'instruction suivante crée un type de donnée *Ch* calqué sur le type *char* :

```
typedef char Ch
```

## Conversion de type de données

On appelle *conversion de type de données* le fait de modifier le type d'une donnée en une autre. Il peut arriver par exemple que l'on veuille travailler sur un type de variable, puis l'utiliser sous un autre type. Imaginons que l'on travaille par exemple sur une variable en virgule flottante (type *float*), il se peut que l'on veuille « supprimer les chiffres après la virgule », c'est-à-dire convertir un *float* en *int*. Cette opération peut être réalisée de deux manières :

- **conversion implicite** : une conversion implicite consiste en une modification du type de donnée effectuée automatiquement par le compilateur. Cela signifie que lorsque l'on va stocker un type de donnée dans une variable déclarée avec un autre type, le compilateur ne retournera pas d'erreur mais effectuera une conversion *implicite* de la donnée avant de l'affecter à la variable.
- `int x;`

```
x = 8.324;
```

x contiendra après affectation la valeur 8.

- **conversion explicite** : une conversion explicite (appelée aussi *opération de cast*) consiste en une modification du type de donnée forcée. Cela signifie que l'on utilise un opérateur dit *de cast* pour spécifier la conversion. L'opérateur de cast est tout simplement le type de donnée, dans lequel on désire convertir une variable, entre des parenthèses précédant la variable.

```
int x;
x = (int)8.324;
```

x contiendra après affectation la valeur 8.

# Les variables du langage C

## Le concept de variable

Une variable est un objet repéré par son nom, pouvant contenir des données, qui pourront être modifiées lors de l'exécution du programme. Les variables en langage C sont typées, c'est-à-dire que les données contenues dans celles-ci possèdent un type, ainsi elles sont donc stockées dans la mémoire et occupent un nombre d'octets dépendant du type de donnée stockée.

En langage C, les noms de variables peuvent être aussi long que l'on désire, toutefois le compilateur ne tiendra compte que des 32 premiers caractères. De plus, elles doivent répondre à certains critères :

- un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un « _ » (pas par un chiffre)
- un nom de variable peut comporter des lettres, des chiffres et le caractère « _ » (les espaces ne sont pas autorisés !)
- les noms de variables ne peuvent pas être les noms suivants (qui sont des noms réservés) :
    - auto
    - break
    - case, char, const, continue
    - default, do, double
    - else, enum, extern
    - float, for
    - goto
    - if, int
    - long
    - register, return
    - short, signed, sizeof, static, struct, switch
    - typedef
    - union, unsigned
    - void, volatile
    - while

| Nom de variable correct | Nom de variable incorrect | Raison |
|---|---|---|
| Variable | Nom de Variable | comporte des espaces |
| Nom_De_Variable | 123Nom_De_Variable | commence par un chiffre |
| nom_de_variable | toto@mailcity.com | caractère spécial @ |
| nom_de_variable_123 | Nom-de-variable | signe - interdit |
| _707 | goto | nom réservé |


Les noms de variables sont sensibles à la casse (le langage C fait la différence entre un nom en majuscules et un nom en minuscules), il faut donc veiller à utiliser des noms comportant la même casse !

## La déclaration de variables

Pour pouvoir utiliser une variable, il faut la définir, c'est-à-dire lui donner un nom, mais surtout un type de donnée à stocker afin qu'un espace mémoire conforme au type de donnée qu'elle contient lui soit réservé.

Une variable se déclare de la façon suivante :

```
type Nom_de_la_variable;
```

ou bien s'il y a plusieurs variables du même type :

```
type Nom_de_la_variable1, Nom_de_la_variable2, ...;
```

## Affectation d'une donnée à une variable

Pour stocker une donnée dans une variable que l'on a initialisée, il faut faire une affectation, c'est-à-dire préciser la donnée qui va être stockée à l'emplacement mémoire qui a été réservé lors de l'initialisation.

Pour cela on utilise l'opérateur d'affectation « = » :

```
Nom_de_la_variable = donnée;
```

Pour stocker le caractère B dans la variable que l'on a appelée *Caractere*, il faudra écrire:

```
Caractere = 'B';
```

Ce qui signifie *stocker la valeur ASCII de « B » dans la variable nommée « Caractere »*. Il est bien évident qu'il faut avoir préalablement déclaré la variable en lui affectant le type *char* :

```
char Caractere;
```

## Initialisation d'une variable

La déclaration d'une variable ne fait que « réserver » un emplacement mémoire où stocker la variable. Tant que l'on ne lui a pas affecté une donnée celle-ci contient ce qui se trouvait précédemment à cet emplacement, que l'on appelle *garbage* (en français *détritus*).

On peut donc affecter une valeur initiale à la variable lors de sa déclaration, on parle alors d'initialisation :

```
type Nom_de_la_variable = donnee;
```

Par exemple :

```
float Toto = 125.36;
```

## Portée (visibilité) des variables

Selon l'endroit où on déclare une variable, celle-ci pourra être accessible (visible) de partout dans le code ou bien que dans une portion confinée de celui-ci (à l'intérieur d'une fonction par exemple), on parle de *portée* (ou *visibilité*) d'une variable.

Lorsqu'une variable est déclarée dans le code même, c'est-à-dire à l'extérieur de toute fonction ou de tout bloc d'instruction, elle est accessible de partout dans le code (n'importe quelle fonction du programme peut faire appel à cette variable). On parle alors de **variable globale**.

Lorsque l'on déclare une variable à l'intérieur d'un bloc d'instructions (entre des accolades), sa portée se confine à l'intérieur du bloc dans lequel elle est déclarée.

- Une variable déclarée au début du code, c'est-à-dire avant tout bloc de donnée, sera globale, on pourra alors l'utiliser à partir de n'importe quel bloc d'instruction.

- Une variable déclarée à l'intérieur d'un bloc d'instructions (dans une fonction ou une boucle par exemple) aura une portée limitée à ce seul bloc d'instruction, c'est-à-dire qu'elle est inutilisable ailleurs, on parle alors de variable locale.

### Définition de constantes

Une constante est une variable dont la valeur est inchangeable lors de l'exécution d'un programme. En langage C, les constantes sont définies grâce à la directive du préprocesseur *#define*, qui permet de remplacer toutes les occurrences du mot qui le suit par la valeur immédiatement derrière elle. Par exemple la directive :

```
#define _Pi 3.1415927
```

remplacera tous les identifiants « _Pi » (sans les guillemets) par la valeur 3.1415927, sauf dans les chaînes de caractères :

```
resultat = _Pi * sin(a); //-> remplacé

resultat = _Pi+1; //-> remplacé

_Pi = 12; //-> remplacé MAIS génère une erreur

resultat = _PiPo; //-> pas remplacé

printf("pi = _Pi "); //-> pas remplacé
```

Toutefois, avec cette méthode les constantes ne sont pas typées, il faut donc utiliser la directive *#define* avec parcimonie...
Il est ainsi préférable d'utiliser le mot clef *const*, qui permet de déclarer des constantes typées :

```
const int dix = 10;
```

De plus, cela permet d'éviter certains problèmes du *#define*, qui fait du « *chercher-remplacer* » textuel sans réfléchir.

## Les opérateurs du langage C

### Qu'est-ce qu'un opérateur ?

Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, etc.
On distingue plusieurs types d'opérateurs :

- les opérateurs de calcul
- les opérateurs d'assignation
- les opérateurs d'incrémentation
- les opérateurs de comparaison
- les opérateurs logiques
- (les opérateurs bit-à-bit)
- (les opérateurs de décalage de bit)

### Les opérateurs de calcul

Les opérateurs de calcul permettent de modifier mathématiquement la valeur d'une variable.

| Opérateur | Dénomination | Effet | Exemple | Résultat (avec x valant 7) |
|---|---|---|---|---|
| + | opérateur d'addition | Ajoute deux valeurs | x+3 | 10 |

| | | | | |
|---|---|---|---|---|
| - | opérateur de soustraction | Soustrait deux valeurs | x-3 | 4 |
| * | opérateur de multiplication | Multiplie deux valeurs | x*3 | 21 |
| / | opérateur de division | Divise deux valeurs | x/3 | 2.3333333 |
| = | opérateur d'affectation | Affecte une valeur à une variable | x=3 | Met la valeur 3 dans la variable x |

## Les opérateurs d'assignation

Ces opérateurs permettent de simplifier des opérations telles que *ajouter une valeur dans une variable et stocker le résultat dans la variable*. Une telle opération s'écrirait habituellement de la façon suivante par exemple : *x=x+2*
Avec les opérateurs d'assignation il est possible d'écrire cette opération sous la forme suivante : *x+=2*
Ainsi, si la valeur de x était 7 avant opération, elle sera de 9 après...

Les autres opérateurs du même type sont les suivants :

| Opérateur | Effet |
|---|---|
| **+=** | additionne deux valeurs et stocke le résultat dans la variable (à gauche) |
| **-=** | soustrait deux valeurs et stocke le résultat dans la variable |
| ***=** | multiplie deux valeurs et stocke le résultat dans la variable |
| **/=** | divise deux valeurs et stocke le résultat dans la variable |

L'opérateur d'affectation = renvoie aussi une valeur, qui est celle de la variable après affectation. Cela permet notamment de faire des affectations en cascade :

```
a = b = c = 1;
```

ce qui correspond à :

```
a = (b = (c = 1));
```

## Les opérateurs d'incrémentation

Ce type d'opérateur permet de facilement augmenter ou diminuer d'une unité une variable. Ces opérateurs sont très utiles pour des structures telles que des boucles, qui ont besoin d'un compteur (variable qui augmente de un en un).

Un opérateur de type *x++* permet de remplacer des notations lourdes telles que *x=x+1* ou bien *x+=1*.

| Opérateur | Dénomination | Effet | Syntaxe | Résultat (avec x valant 7) |
|---|---|---|---|---|
| **++** | Incrémentation | Augmente d'une unité la variable | x++ | 8 |
| **--** | Décrémentation | Diminue d'une unité la variable | x-- | 6 |

## Les opérateurs de comparaison

| Opérateur | Dénomination | Effet | Exemple | Résultat (avec x valant 7) |
|---|---|---|---|---|
| == **A ne pas confondre avec le signe d'affectation (=) !** | opérateur d'égalité | Compare deux valeurs et vérifie leur égalité | x==3 | Retourne 1 si x est égal à 3, sinon 0 |
| < | opérateur d'infériorité stricte | Vérifie qu'une variable est strictement inférieure à une valeur | x<3 | Retourne 1 si x est inférieur à 3, sinon 0 |
| <= | opérateur d'infériorité | Vérifie qu'une variable est inférieure ou égale à une valeur | x<=3 | Retourne 1 si x est inférieur ou égal à 3, sinon 0 |

| | | | | |
|---|---|---|---|---|
| > | opérateur de supériorité stricte | Vérifie qu'une variable est strictement supérieure à une valeur | x>3 | Retourne 1 si x est supérieur à 3, sinon 0 |
| >= | opérateur de supériorité | Vérifie qu'une variable est supérieure ou égale à une valeur | x>=3 | Retourne 1 si x est supérieur ou égal à 3, sinon 0 |
| != | opérateur de différence | Vérifie qu'une variable est différente d'une valeur | x!=3 | Retourne 1 si x est différent de 3, sinon 0 |

## Les opérateurs logiques (booléens)

Ce type d'opérateur permet de vérifier si plusieurs conditions sont vraies :

| Opérateur | Dénomination | Effet | Syntaxe |
|---|---|---|---|
| **\|\|** | OU logique | Vérifie qu'une des conditions est réalisée | ((condition1)\|\|(condition2)) |
| **&&** | ET logique | Vérifie que toutes les conditions sont réalisées | ((condition1)&&(condition2)) |
| **!** | NON logique | Inverse l'état d'une variable booléenne (retourne la valeur 1 si la variable vaut 0, 0 si elle vaut 1) | (!condition) |

## (Les opérateurs bit-à-bit)

Si vous ne comprenez pas ces opérateurs cela n'est pas important, vous n'en aurez probablement pas l'utilité. Pour ceux qui voudraient comprendre, rendez-vous aux chapitres suivants :

- Compréhension du binaire
- Représentation des données
- Instructions arithmétiques et logiques en assembleur

Ce type d'opérateur traite ses opérandes comme des données binaires, plutôt que des données décimales, hexadécimales ou octales. Ces opérateurs traitent ces données selon leur représentation binaire mais retournent des valeurs numériques standard dans leur format d'origine.

Les opérateurs suivants effectuent des opérations bit-à-bit, c'est-à-dire avec des bits de même poids.

| Opérateur | Dénomination | Effet | Syntaxe | Résultat |
|---|---|---|---|---|
| **&** | ET bit-à-bit | Retourne 1 si les deux bits de même poids sont à 1 | 9 & 12 (1001 & 1100) | 8 (1000) |
| **\|** | OU bit-à-bit | Retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux) | 9 \| 12 (1001 \| 1100) | 13 (1101) |
| **^** | OU bit-à-bit exclusif | Retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux) | 9 ^ 12 (1001 ^ 1100) | 5 (0101) |

## (Les opérateurs de décalage de bit)

Si vous ne comprenez pas ces opérateurs cela n'est pas important, vous n'en aurez probablement pas l'utilité. Pour ceux qui voudraient comprendre, rendez-vous aux chapitres suivants :

- Compréhension du binaire
- Représentation des données
- Instructions arithmétiques et logiques en assembleur

Ce type d'opérateur traite ses opérandes comme des données binaires d'une longueur de 32 bits, plutôt que des données décimales, hexadécimales ou octales. Ces opérateurs traitent ces données selon leur représentation binaire mais retournent des valeurs numériques standard dans leur format d'origine.

Les opérateurs suivants effectuent des décalages sur les bits, c'est-à-dire qu'ils décalent chacun des bits d'un nombre de positions vers la gauche ou vers la droite. La première opérande désigne la donnée sur laquelle on va faire le décalage, la seconde désigne le nombre de décalages.

| Opérateur | Dénomination | Effet | Syntaxe | Résultat |
|---|---|---|---|---|
| **<<** | Décalage à gauche | Décale les bits vers la gauche (multiplie par 2 à chaque décalage). Les zéros qui sortent à gauche sont perdus, tandis que des zéros sont insérés à droite | 6 << 1 (110 << 1) | 12 (1100) |
| **>>** | Décalage à droite avec conservation du signe | Décale les bits vers la droite (divise par 2 à chaque décalage). Les zéros qui sortent à droite sont perdus, tandis que le bit non nul de poids plus fort est recopié à gauche | 6 >> 1 (0110 >> 1) | 3 (0011) |

### Les priorités

Lorsque l'on associe plusieurs opérateurs, il faut que le navigateur sache dans quel ordre les traiter, voici donc dans **l'ordre décroissant** les priorités de tous les opérateurs :

<table>
<tr><th colspan="12">Priorité des opérateurs</th></tr>
<tr><td></td><td>()</td><td>[]</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>--</td><td>++</td><td>!</td><td>~</td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>*</td><td>/</td><td>%</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>+</td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td><<</td><td>>></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td><</td><td><=</td><td>>=</td><td>></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>==</td><td>!=</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>&</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>^</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>|</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>&&</td><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>?:</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>=</td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td><<=</td><td>>>=</td><td>&=</td><td>^=</td><td>|=</td></tr>
</table>

## Les structures conditionnelles

### Qu'est-ce qu'une structure conditionnelle ?

On appelle ***structure conditionnelle*** les instructions qui permettent de tester si une condition est vraie ou non. Ces structures conditionnelles peuvent être associées à des structures qui se répètent suivant la réalisation de la condition, on appelle ces structures des ***structures de boucle***.

### La notion de bloc

Une expression suivie d'un point-virgule est appelée instruction. Voici un exemple d'instruction :

```
a++;
```

Lorsque l'on veut regrouper plusieurs instructions, on peut créer ce que l'on appelle un **bloc**, c'est-à-dire un ensemble d'instructions (suivies respectivement par des points-virgules) et comprises entre les accolades { et }.

Les instructions *if*, *while* et *for* peuvent par exemple être suivies d'un bloc d'instructions à exécuter...

## L'instruction if

L'instruction if est la structure de test la plus basique, on la retrouve dans tous les langages (avec une syntaxe différente...). Elle permet d'exécuter une série d'instructions si jamais une condition est réalisée.

La syntaxe de cette expression est la suivante :

```
if (condition réalisée) {

   liste d'instructions;

}
```

**Remarques :**

- la condition doit être entre des parenthèses
- il est possible de définir plusieurs conditions à remplir avec les opérateurs *ET* et *OU* (&& et ||)
  Par exemple l'instruction suivante teste si les deux conditions sont vraies :

  ```
  if ((condition1)&&(condition2))
  ```

  L'instruction suivante exécutera les instructions si l'une ou l'autre des deux conditions est vraie :

  ```
  if ((condition1)||(condition2))
  ```

- s'il n'y a qu'une instruction, les accolades ne sont pas nécessaires...
- les instructions situées dans le bloc qui suit *else* sont les instructions qui seront exécutées si la ou les conditions ne sont pas remplies

## L'instruction if ... else

L'instruction *if* dans sa forme basique ne permet de tester qu'une condition, or la plupart du temps on aimerait pouvoir choisir les instructions à exécuter **en cas de non réalisation de la condition**...
L'expression *if ... else* permet d'exécuter une autre série d'instructions en cas de non-réalisation de la condition.

La syntaxe de cette expression est la suivante :

```
if (condition réalisée) {

   liste d'instructions

}

else {
```

```
    autre série d'instructions
}
```

## Une façon plus courte de faire un test

Il est possible de faire un test avec une structure beaucoup moins lourde grâce à la structure suivante :

```
(condition) ? instruction si vrai : instruction si faux
```

**Remarques :**

- la condition doit être entre des parenthèses
- Lorsque la condition est vraie, l'instruction de gauche est exécutée
- Lorsque la condition est fausse, l'instruction de droite est exécutée
- En plus d'être exécutée, la structure *?:* renvoie la valeur résultant de l'instruction exécutée. Ainsi, cette forme *?:* est souvent utilisée comme suit :
-
- `position = ((enAvant == 1) ? compteur+1 : compteur-1);`
-

## L'instruction switch

L'instruction *switch* permet de faire plusieurs tests de valeurs sur le contenu d'une même variable. Ce branchement conditionnel simplifie beaucoup le test de plusieurs valeurs d'une variable, car cette opération aurait été compliquée (mais possible) avec des *if* imbriqués. Sa syntaxe est la suivante :

```
switch (Variable) {
case Valeur1:
        Liste d'instructions;
        break;
case Valeur2:
        Liste d'instructions;
        break;
case Valeurs...:
        Liste d'instructions;
        break;
default:
        Liste d'instructions;
}
```

Les parenthèses qui suivent le mot clé *switch* indiquent une expression dont la valeur est testée successivement par chacun des *case*. Lorsque l'expression testée est égale à une des valeurs suivant un *case*, la liste d'instructions qui suit celui-ci est exécutée. Le mot clé *break* indique la sortie de la structure conditionnelle. Le mot clé *default* précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs.

N'oubliez pas d'insérer des instructions *break* entre chaque test, ce genre d'oubli est difficile à détecter car aucune erreur n'est signalée...
En effet, lorsque l'on omet le *break*, l'exécution continue dans les blocs suivants !

Cet état de fait peut d'ailleurs être utilisé judicieusement afin de faire exécuter les mêmes instructions pour différentes valeurs consécutives, on peut ainsi mettre plusieurs cases avant le bloc :

```
switch(variable)

{

case 1:

case 2:

{ instructions exécutées pour variable = 1 ou pour variable =
2 }

break;

case 3:

{ instructions exécutées pour variable = 3 uniquement }

break;

default:

{ instructions exécutées pour toute autre valeur de variable }

}
```

## Les boucles

Les boucles sont des structures qui permettent d'exécuter plusieurs fois la même série d'instructions jusqu'à ce qu'une condition ne soit plus réalisée...
On appelle parfois ces structures *instructions répétitives* ou bien *itérations*.
La façon la plus commune de faire une boucle est de créer un compteur (une variable qui s'incrémente, c'est-à-dire qui augmente de 1 à chaque tour de boucle) et de faire arrêter la boucle lorsque le compteur dépasse une certaine valeur.

## La boucle for

L'instruction *for* permet d'exécuter plusieurs fois la même série d'instructions : c'est une boucle !

Dans sa syntaxe, il suffit de préciser le nom de la variable qui sert de compteur (et éventuellement sa valeur de départ, la condition sur la variable pour laquelle la boucle s'arrête (basiquement une condition qui teste si la valeur du compteur dépasse une limite) et enfin une instruction qui incrémente (ou décrémente) le compteur.

La syntaxe de cette expression est la suivante :

```
for (compteur; condition; modification du compteur) {

    liste d'instructions;

}
```

Par exemple :

```
for (i=1; i<6; i++) {

    printf("%d", i);

}
```

Cette boucle affiche 5 fois la valeur de *i*, c'est-à-dire 1, 2, 3, 4, 5.
Elle commence à i=1, vérifie que i est bien inférieur à 6, etc. jusqu'à atteindre la valeur i=6, pour laquelle la condition ne sera plus réalisée, la boucle s'interrompra et le programme continuera son cours.

## L'instruction while



- il faudra toujours vérifier que la boucle a bien une condition de sortie (i.e. le compteur s'incrémente correctement)
- une instruction *printf();* dans votre boucle est un bon moyen pour vérifier la valeur du compteur pas à pas en l'affichant !
- il faut bien compter le nombre de fois que l'on veut faire exécuter la boucle :
  - *for(i=0;i<10;i++)* exécute 10 fois la boucle (i de 0 à 9)
  - *for(i=0;i<=10;i++)* exécute 11 fois la boucle (i de 0 à 10)
  - *for(i=1;i<10;i++)* exécute 9 fois la boucle (i de 1 à 9)
  - *for(i=1;i<=10;i++)* exécute 10 fois la boucle (i de 1 à 10)

L'instruction *while* représente un autre moyen d'exécuter plusieurs fois la même série d'instructions.

La syntaxe de cette expression est la suivante :

```
while (condition réalisée) {

      liste d'instructions;

}
```

Cette instruction exécute la liste d'instructions **tant que** (*while* est un mot anglais qui signifie *tant que*) la condition est réalisée.



La condition de sortie pouvant être n'importe quelle structure conditionnelle, les risques de boucle infinie (boucle dont la condition est toujours vraie) sont grands, c'est-à-dire qu'elle risque de provoquer un plantage du programme en cours d'exécution !

## Saut inconditionnel

Il peut être nécessaire de faire sauter à la boucle une ou plusieurs valeurs sans pour autant mettre fin à celle-ci.

La syntaxe de cette expression est « *continue;* » (cette instruction se place dans une boucle !), on l'associe généralement à une structure conditionnelle, sinon les lignes situées entre cette instruction et la fin de la boucle seraient obsolètes.

Exemple : Imaginons que l'on veuille imprimer pour x allant de 1 à 10 la valeur de 1/(x-7) ; il est évident que pour x=7 il y aura une erreur. Heureusement, grâce à l'instruction *continue* il est possible de traiter cette valeur à part puis de continuer la boucle !

```
x=1;

while (x<=10) {

      if (x == 7) {

         printf("Division par zéro !");

         continue;

      }

      a = 1/(x-7);

      printf("%f", a);

      x++;

}
```

Il y avait une erreur dans ce programme... peut-être ne l'avez-vous pas vue :
Lorsque x est égal à 7, le compteur ne s'incrémente plus, il reste constamment à la valeur 7, il aurait fallu écrire :

```
x=1;

while (x<=10) {

      if (x == 7) {

         printf("Division par 0");

         x++;
```

```
            continue;
        }
        a = 1/(x-7);
        printf("%f", a);
        x++;
}
```

### Arrêt inconditionnel

A l'inverse, il peut être voulu d'arrêter prématurément la boucle, pour une autre condition que celle précisée dans l'en-tête de la boucle. L'instruction *break* permet d'arrêter une boucle (*for* ou bien *while*). Il s'agit, tout comme *continue*, de l'associer à une structure conditionnelle, sans laquelle la boucle ne ferait jamais plus d'un tour !

Dans l'exemple de tout à l'heure, par exemple si l'on ne savait pas à quel moment le dénominateur (x-7) s'annule (bon... OK... pour des équations plus compliquées par exemple) il serait possible de faire arrêter la boucle en cas d'annulation du dénominateur, pour éviter une division par zéro !

```
for (x=1; x<=10; x++) {
    a = x-7;
    if (a == 0) {
       printf("Division par 0");
       break;
    }
    printf("%f", 1/a);
}
```

# Les tableaux

### Type de données complexes

Les variables, telles que nous les avons vues, ne permettent de stocker qu'une seule donnée à la fois. Or, pour de nombreuses données, comme cela est souvent le cas, des variables distinctes seraient beaucoup trop lourdes à gérer. Heureusement, le langage C propose des structures de données permettant de stocker l'ensemble de ces données dans une « variable commune ». Ainsi, pour accéder à ces valeurs il suffit de parcourir la variable de type complexe composée de « variables » de type simple.

Le langage C propose deux types de structures :

- **les tableaux**, permettant de stocker plusieurs données de même type
- **les structures**, pouvant contenir des données hétérogènes

### La notion de tableau

On appelle *tableau* une variable composée de données de même type, stockée de manière contiguë en mémoire (les unes à la suite des autres).

Un tableau est donc une suite de cases (espace mémoire) de même taille. La taille de chacune des cases est conditionnée par le type de donnée que le tableau contient.
Les éléments du tableau peuvent être :

- des données de type simple : int, char, float, long, double...
(la taille d'une case du tableau est alors le nombre d'octets sur lequel la donnée est codée)
- des pointeurs (objets contenant une adresse mémoire. Ce type d'entité sera expliqué dans les chapitres suivants)
- des tableaux
- des structures

Voici donc une manière de représenter un tableau :

| | | | | | | |
|---|---|---|---|---|---|---|
| donnée | donnée | donnée | ... | donnée | donnée | donnée |

- Lorsque le tableau est composé de données de type simple, on parle de *tableau monodimensionnel* (ou *vecteur*)
- Lorsque celui-ci contient lui-même d'autres tableaux on parle alors de *tableaux multidimensionnels* (aussi *matrice* ou *table*)

## Les tableaux unidimensionnels

### Déclaration

Un tableau unidimensionnel est un tableau qui contient des éléments simples (des éléments qui ne sont pas des tableaux). Un tableau unidimensionnel est donc une suite de « cases » de même taille contenant des éléments d'un type donné (de la longueur de la case en quelque sorte).
Un tableau contenant des entiers peut se représenter de la façon suivante :

| | | | | | | |
|---|---|---|---|---|---|---|
| int | int | int | ... | int | int | int |

En langage C, la syntaxe de la définition d'un tableau unidimensionnel est la suivante :

```
type Nom_du_tableau [Nombre d'éléments]
```

- *type* définit le type d'élément que contient le tableau (rappel : un tableau en langage C est composé uniquement d'éléments de même type), c'est-à-dire qu'il définit la taille d'une case du tableau en mémoire
- *Nom_du_tableau* est le nom que l'on décide de donner au tableau, le nom du tableau suit les mêmes règles qu'un nom de variable
- *Nombre d'éléments* est un nombre entier qui détermine le nombre de cases que le tableau doit comporter

Voici par exemple la définition d'un tableau qui doit contenir 8 éléments de type char :

```
char Tableau [8]
```

### Calcul de la taille du tableau

Etant donné qu'un tableau est composé d'un nombre fixé d'éléments d'un type donné, la taille d'un tableau est déterminée dès sa définition.

Pour connaître la taille d'un tableau, c'est-à-dire déterminer le nombre d'octets que celui-ci occupe en mémoire, il y a deux possibilités :

- **Calculer manuellement la taille du tableau** : il suffit de multiplier la taille du type d'élément par le nombre d'éléments qui composent le tableau
- **Utiliser l'opérateur *sizeof()*** : l'opérateur *sizeof()* permet de retourner directement la taille de l'élément qui lui est passé en argument, ainsi en lui passant un tableau comme opérande, *sizeof()* est capable de vous retourner directement la taille de celui-ci

Voici différents exemples de tableaux, et leurs tailles respectives :

| Définition du tableau | Taille du tableau (en octets) |
|---|---|
| char Tableau1[12] | 1 * 12 = 12 |
| int Tableau2[10] | 2 * 10 = 20 |
| float Tableau3[8] | 4 * 8 = 32 |
| double Tableau4[15] | 8 * 15 = 120 |

### Accéder aux éléments

Pour accéder à un élément du tableau, le nom que l'on a donné à celui-ci ne suffit pas car il comporte plusieurs éléments. Ainsi, on définit un nombre appelé **indice** (en anglais *index*) qui, combiné avec le nom du tableau, permet de décrire exactement chaque élément.

Pour accéder à un élément du tableau, il suffit donc de donner le nom du tableau, suivi de l'indice de l'élément entre crochets :

```
Nom_du_tableau[indice]
```



- L'indice du premier élément du tableau est **0**
- Un indice est toujours positif
- L'indice du dernier élément du tableau est égal au nombre d'éléments - 1

Ainsi, on accédera au 5ème élément du tableau en écrivant :

```
Nom_du_tableau[4]
```

## Manipuler les éléments

Un élément du tableau (repéré par le nom du tableau et son indice) peut être manipulé exactement comme une variable, on peut donc effectuer des opérations avec (ou sur) des éléments de tableau.

Définissons un tableau de 10 entiers :

```
int Toto[10];
```

Pour affecter la valeur 6 au huitième élément on écrira :

```
Toto[7] = 6;
```

Pour affecter au 10ème élément le résultat de l'addition des éléments 1 et 2, on écrira :

```
Toto[9] = Toto[0] + Toto[1];
```

## Initialiser les éléments

Lorsque l'on définit un tableau, les valeurs des éléments qu'il contient ne sont pas définies, il faut donc les initialiser, c'est-à-dire leur affecter une valeur.
Une méthode rustique consiste à affecter des valeurs aux éléments un par un :

```
Toto[0] = Toto[1] = Toto[2] = 0;
```

L'intérêt de l'utilisation d'un tableau est alors bien maigre...

Une manière plus élégante consiste à utiliser le fait que pour passer d'un élément du tableau à l'élément suivant il suffit d'incrémenter son indice. Il est donc possible d'utiliser une boucle qui va permettre d'initialiser successivement chacun des éléments grâce à un compteur qui servira d'indice :

```
int Toto[10];

int Indice;

for (Indice = 0; Indice <= 9; Indice++) {

Toto[Indice] = 0;

}
```

Cette méthode, aussi utile soit elle, n'a d'intérêt que lorsque les éléments du tableau doivent être initialisés à une valeur unique ou une valeur logique (proportionnelle à l'indice par exemple). Pour initialiser un tableau avec des valeurs spécifiques, il est possible d'initialiser le tableau à la définition en plaçant entre accolades les valeurs, séparées par des virgules :

```
int Toto[10] = {1, 2, 6, 5, 2, 1, 9, 8, 1, 5};
```



- Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau
- Les valeurs entre accolades doivent être des constantes (l'utilisation de variables provoquera une erreur du compilateur)
- Si le nombre de valeurs entre accolades est inférieur au nombre d'éléments du tableau, les derniers éléments sont initialisés à 0
- Il doit y avoir au moins une valeur entre accolades

Ainsi, l'instruction suivante permet d'initialiser tous les éléments du tableau à zéro :

```
int Toto[10] = {0};
```

Il est conseillé d'employer le plus possible des constantes dans vos programmes, notamment pour la taille des tableaux. Le code ci-dessus peut s'écrire ainsi :

```
#define NB_ELEMENT_TOTO 10

int Toto[NB_ELEMENT_TOTO];

int Indice;

for (Indice = 0; Indice < NB_ELEMENT_TOTO; Indice++) {

Toto[Indice] = 0;

}
```

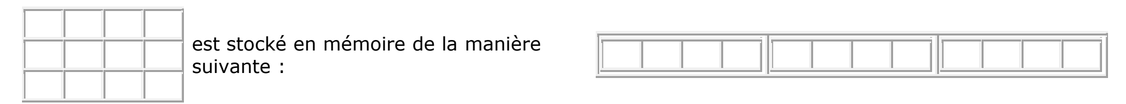
Voici les avantages liés à l'utilisation de constantes :

- Moins d'erreurs d'exécution dues à un débordement difficile à déceler.
- Pour modifier la taille du tableau il suffit de changer le define en début du code source.
- Le code possède une lisibilité accrue.

## Les tableaux multidimensionnels

Les tableaux multidimensionnels sont des tableaux qui contiennent des tableaux.

Par exemple le tableau bidimensionnel (3 lignes, 4 colonnes) suivant, est en fait un tableau comportant 3 éléments, chacun d'entre eux étant un tableau de 4 éléments :



est stocké en mémoire de la manière suivante :

### Définition

Un tableau multidimensionnel se définit de la manière suivante :

```
type Nom_du_tableau [a1][a2][a3] ... [aN]
```

- Chaque élément entre crochets désigne le nombre d'éléments dans chaque dimension
- Le nombre de dimensions n'est pas limité

Un tableau d'entiers positifs à deux dimensions (3 lignes, 4 colonnes) se définira avec la syntaxe suivante :

```
int Tableau [3][4]
```

On peut représenter un tel tableau de la manière suivante :

| | | | |
|---|---|---|---|
| Tableau[0][0] | Tableau[0][1] | Tableau[0][2] | Tableau[0][3] |
| Tableau[1][0] | Tableau[1][1] | Tableau[1][2] | Tableau[1][3] |
| Tableau[2][0] | Tableau[2][1] | Tableau[2][2] | Tableau[2][3] |

Il va de soi que cette représentation est arbitraire, car elle suppose que le premier indice est l'indice de ligne, et le second est l'indice de colonne.
On aurait tout aussi bien pu représenter le tableau de la manière suivante :

| | | |
|---|---|---|
| Tableau[0][0] | Tableau[1][0] | Tableau[2][0] |
| Tableau[0][1] | Tableau[1][1] | Tableau[2][1] |
| Tableau[0][2] | Tableau[1][2] | Tableau[2][2] |
| Tableau[0][3] | Tableau[1][3] | Tableau[2][3] |

On utilise toutefois généralement la première représentation, car elle correspond mieux à la façon selon laquelle le tableau est stocké en mémoire.

### Initialiser les éléments

L'initialisation d'un tableau multidimensionnel se fait à peu près de la même façon que pour les tableaux unidimensionnels. Il y a donc plusieurs façons d'initialiser un tableau multidimensionnel :

- **Initialisation individuelle de chaque élément** :
- `Nom_du_tableau [0][0] = 2;`
- 
- `Nom_du_tableau [0][1] = 3;`
- 

```
...
```

- **Initialisation grâce à des boucles** :
  Il faut faire des boucles imbriquées correspondant chacune à un indice d'une dimension. Par exemple les éléments de Tableau[3][4] pourront être initialisés à 0 par les instructions suivantes :

```
int i,j;

for (i=0; i<=2; i++){

  for (j=0; j<=3; j++){

            Tableau[i][j] = 0;

  }

}
```

- **Initialisation à la définition** :

```
type Nom_du_tableau [Taille1][Taille2]...[TailleN] = {a1, a2, ... aN};
```

Les valeurs sont attribuées aux éléments successifs en incrémentant d'abord les indices de droite, c'est-à-dire pour un tableau à 2 dimensions : [0][0], [0][1], [0][2] ... puis [1][0] etc.

- **Initialisation grâce à la fonction *memset*** :

```
int toto[10];

/* Met à zéro toute la zone mémoire */

memset(*toto, 0, sizeof(toto));

Synopsis de la fonction memset :

#include <string.h>

void * memset ( void * buffer, int c, size_t num );
```

# Les chaînes de caractère

## Qu'est-ce qu'une chaîne de caractères ?

Une chaîne de caractères (appelée *string* en anglais) est une suite de caractères, c'est-à-dire un ensemble de symboles faisant partie du jeu de caractères, défini par le code ASCII. En langage C, une chaîne de caractères est un tableau, comportant plusieurs données de type **char**, dont le dernier élément est le caractère nul '\0', c'est-à-dire le premier caractère du code ASCII (dont la valeur est 0).
Ce caractère est un caractère de contrôle (donc non affichable) qui permet d'indiquer une fin de chaîne de caractères. Ainsi une chaîne composée de *n* éléments sera en fait un tableau de *n+1* éléments de type char.
On peut par exemple représenter la chaîne « Bonjour » de la manière suivante :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | o | n | j | o | u | r | \0 |

## Créer une chaîne de caractères

Pour définir une chaîne de caractères en langage C, il suffit de définir un tableau de caractères. Le nombre maximum de caractères que comportera la chaîne sera égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne).

```
char Nom_du_tableau[Nombre_d_elements]
```



- Le nombre d'éléments que comporte le tableau définit la taille maximale de la chaîne, on peut toutefois utiliser partiellement cet espace en insérant le caractère de fin de chaîne à l'emplacement désiré dans le tableau.

**Astuce !** En définissant le tableau de la manière suivante, vous mettez en évidence le nombre de caractères maximal de la chaîne :

```
char Nom_du_tableau[Nombre_d_elements + 1]
```

Par exemple :

```
char Chaine[50 + 1]
```

## Initialiser une chaîne de caractères

Comme généralement en langage C, il faut initialiser votre chaîne de caractères, c'est-à-dire remplir les cases du tableau avec des caractères, sachant que celui-ci devra **obligatoirement** contenir le caractère de fin de chaîne '\0'. Il y a deux façons de procéder :

- remplir manuellement le tableau case par case
- utiliser les fonctions de manipulation de chaînes fournies dans les librairies standard

Voici un exemple d'initialisation manuelle de chaîne de caractères :

```
#include <stdio.h>

void main(){

          char Chaine[20+1];

          Chaine[0]= 'B';

          Chaine[1]= 'o';

          Chaine[2]= 'n';

          Chaine[3]= 'j';

          Chaine[4]= 'o';

          Chaine[5]= 'u';

          Chaine[6]= 'r';

          Chaine[7]= '\0';

}
```

Voici une autre façon (plus simple) d'initialiser une chaîne de caractères :

```
#include <stdio.h>

void main(){

          char Chaine[20+1]={ 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };

}
```

## Les fonctions de manipulation de chaînes de caractères

De nombreuses fonctions de manipulation de chaînes sont directement fournies. Ces fonctions se trouvent dans le fichier d'en-tête <string.h>, c'est la raison pour laquelle il faut ajouter la ligne suivante en début de programme :

```
#include <string.h>
```

Le fichier *<string.h>* contient les prototypes de nombreuses fonctions permettant de simplifier l'utilisation et la manipulation de chaînes (environ une quarantaine).

# Introduction Langage C

## 1/ Historique

Langage a la fois de haut niveau et bas niveau crée en 1970 pas BW Kernigham et D.M Ritchie pour le développement du system UNIX (avant le C, le langage B ➔ laboratoire Bell+ université américaine).
-Haut niveau :
C'est un langage structuré (sous programme, boucles, si, variables, constantes …)

Bas niveau :
Opérateur permettant de manipuler des instructions et les données (variable) au niveau du langage machine c'est-à-dire BINAIRE.

C'est un langage faiblement typé : ce n'est pas le type qui est important mais la place en octet qu'il occupe.

Pourquoi le C ?

haut et bas niveau
Plusieurs niveaux (Linux, Unix, Windows…)
D'autres langages découlent du C : c++, java, PHP…

2/ Etapes de programmation : du problème jusqu'à l'exécution du programme.

analyse descendante
présentation des ressources (variables)
algorithme
vérifier que l'algo fonctionne
traduction dans un langage de haut niveau ex : « c »
Compilation du programme
Exécution du programme.

3/ compilation et édition de liens (exécution) d'un programme
La compilation : elle vérifie si la syntaxe du programme est correcte.
(Voir cours feuille).

4/ Composition d'un programme type en C

4.1. Créer une bibliothèque personnelle (nom biblio.h)

```
#ifndef Biblio              //Si la bibliothèque biblio n'existe pas
#define biblio              // Alors definir cette bibliothèque BIBLIO
#include  <stdio.h>                  // inclure bibliothèque des fonctions
// D'entrée-sortie standard
#inclue <stdlib.h>          // inclure bibliothèque des fonction systèmes
#endif                      // fin du #ifndef
```

Ce qui commence # directive de pré compilateur (avant compilation)

Tous les mots réserve en C sont en minuscules. Le C fait la différence entre le minuscule et majuscule.
// = commentaire mais aussi /* Commentaire */
« BIBLIO » nom symbolique : comme les variable et les constantes les noms commencent par 1 lettre ou _
<…> bibliothèque standard.

4.2. Dans un fichier source « .C »

#include « nom biblio.h » // inclure sa bibliothèque personnelle
// Qui se trouve dans le même dossier

Exemple convertisseur Euro → franc

#define taux 6.55957 //déclarer une constante taux= 6.55957

Int main( ) // toujours 1 main : une commande principal

{ //début du bloc main

Float (réel) franc, euro ; // mettre le type suivi des noms des variables

Do (répéter) {
Afficher ←Printf (« entrer le montant en franc : \n ») ; // \n = a la ligne

Lire ←Scanf (« %f », & Franc) ; // %f format de saisie du float, &adresse de la //variable franc
} While (Franc<0) ;

Euro = franc / taux ; //convertir franc en euro

Printf (« montant Euro = %.2f \n », euro) ; // Afficher le montant en euro

// Faire une pause

System (« pause ») ; //fonction système

Return 0 ; // retourner une valeur de succès 0

} //fin du main

CODE JUSTE

```
#include "BIBLIO.h"

#include "BIBLIO.h "

#define taux 6.55957

int main ()
{


Float Franc, euro ;

Do {

        Printf (" entrer le montant en franc :\n ");

        Scanf (" %f ", & Franc);
} While (Franc<0) ;

Euro = Franc / taux ;
Printf (" montant Euro = %.2f \n ", euro) ;


System (" pause ") ;
Return 0 ;
}
```

5) Les constantes

2 manières :
- 1ere manières

```
#define nom valeur
```

Ex #define PI 3.14   Dans ce cas c’est 1 macro définition : le processeur remplace  toute occurrence de PI par 3.14

Universelle a tous les compilateurs ➔ aucune place  mémoire reservée pour la constante => adresse inaccessible.

2eme manière l'intérieur du main

```
Const type nom =3.14 ;
Ex :
Int main ()
{Const float PI=3.14 ;
}
```

Avantage : place mémoire réservé
Inconvénient pas universel (ex dev c++)

6) Les types :

Pour un type en algo il y a plusieurs types possibles en C
Rappel : le c est 1 langage faiblement typé : le type en donne le nombre d'octets en mémoire de ce type

ENTIER

- Int : entier sur 4 octets de -2 ^31 a 2^31 -1

unsigned Int : entier >= 0 sur 4octets de 0 à 2^32 – 1
Short : entier sur 2 octets de -2^15 à 2^15-1
Unsigned short : > = 0 sur 2 octets de 0 a 2^16-1
Long : entier sur 8 octets de -2^63 a 2^63 -1
Unsigned long : entier > = 0 8 octets de 0 a 2^64 -1

REEL

float : réel sur 4 octets avec exposant et mantisse
Double : réel 8octets

CARACTERE : ➔ code ASCII = code numérique associé a chaque touche du clavier (caractère)

Char : 1 octet compatible avec les entiers
Unsigned char : idéal pour simuler le type booléen

7) les variables

Type nomvar1, …. , nomvarN ;

Ex : 2 variable X et Y Y de type entier
3 variables riri, fifi, loulou, de type caractère
1 variable ok de type booléen

```
Int X, Y ;
Char riri, fifi, loulou ;
```

Unsigned char ok ;

Remarque : nom des variables même règle que les constantes.

8) les Opérateurs.

8.1 L'affectation

Nomvariable = valeur ; // mettre 1 valeur dans le 1 variable

Ex :

X=5 ;
Y=x *3

8.2 Les opérateur de comparaison

> = = pour l'égalité
<
<=
> = != pour différent

8.3 Opérateur arithmétique
+ - / * pour les entiers et réels
% (modulo) par les entiers ex 5%2=1

Opérateurs booléens

Ou ||
Et &&
No !

9) les entrées/ sorties formatées

Le langage c offre la possibilité de réaliser des affichages formaté de teste et de contenu de variables. Ces notions de format sont également exploitées lors de la lecture de valeurs au clavier

9.1 Les formats : a chaque type C est associé un format

%d pour int %c pour char
%f pour float %ld pour long
%fl pour double

9.2 Printf (affichage) bibliothèque <stdio.h>
Int X ;

Float Y ;

X=2 ;
Y=2.7 ;

//afficher les valeurs de X et X
Printing (« les valeurs de X =%d, Y=%f \n », X, Y) ;

9.3 Scanf (lire)

Ne pas oublier de mettre l'adresse (&) des variables

Exemple : Lire X , Y

Scanie (« %d%f », &X, &Y) ;

10) structure conditionnel if (si)

Exemple : lire une age, s'il est <25 ou >60 accorder 1 réduction

```
#include <stdio.h> //bibliothèque entrée-sortie standard
#include <stdlib.h> //bibliothèque système

Int main() //programme principal

{ //début
     Int age ; //variable
    Printf (« entrez votre age : ») ;
    Scanf (« %d », &age »);
    If (age < 25 || age>60°)
    {Printf (« réduction accorde \n »)}; // les {falcutative quand
une instruction
    Else
    {Printf ('réduction refusée \n ») ;}
    System (« pause ») ; return 0 ;
}
```